8

Collections

WHAT'S IN THIS CHAPTER?

- Understanding collection interfaces and types
- Working with lists, queues, and stacks
- Working with linked and sorted lists
- Using dictionaries and sets
- Evaluating performance
- Using immutable collections

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/Collections`.

The code for this chapter is divided into the following major examples:

- ListSamples
- QueueSample
- LinkedListSample
- SortedListSample
- DictionarySample
- SetSample
- ImmutableCollectionsSample

All the projects have nullable reference types enabled.

OVERVIEW

Chapter 6, "Arrays," covers arrays and the interfaces implemented by the `Array` class. The size of arrays is fixed. If the number of elements is dynamic, you should use a collection class instead of an array.

`List<T>` is a collection class that can be compared to arrays, but there are also other kinds of collections: queues, stacks, linked lists, dictionaries, and sets. The other collection classes have partly different APIs to access the elements in the collection and often a different internal structure for how the items are stored in memory. This chapter covers all of these collection classes and their differences, including performance differences.

COLLECTION INTERFACES AND TYPES

Most collection classes are in the `System.Collections` and `System.Collections.Generic` namespaces. Generic collection classes are located in the `System.Collections.Generic` namespace. Collection classes that are specialized for a specific type are located in the `System.Collections.Specialized` namespace. Thread-safe collection classes are in the `System.Collections.Concurrent` namespace. Immutable collection classes are in the `System.Collections.Immutable` namespace.

Of course, there are also other ways to group collection classes. Collections can be grouped into lists, collections, and dictionaries based on the interfaces that are implemented by the collection class.

> **NOTE** *You can read detailed information about the interfaces* `IEnumerable` *and* `IEnumerator` *in Chapter 6.*

The following table describes the most important interfaces implemented by collections and lists:

INTERFACE	DESCRIPTION
`IEnumerable<T>`	The interface `IEnumerable` is required by the `foreach` statement. This interface defines the method `GetEnumerator`, which returns an enumerator that implements the `IEnumerator` interface.
`ICollection<T>`	`ICollection<T>` is implemented by generic collection classes. With this you can get the number of items in the collection (`Count` property) and copy the collection to an array (`CopyTo` method). You can also add and remove items from the collection (`Add`, `Remove`, `Clear`).
`IList<T>`	The `IList<T>` interface is for lists where elements can be accessed from their position. This interface defines an indexer, as well as ways to insert or remove items from specific positions (`Insert`, `RemoveAt` methods). `IList<T>` derives from `ICollection<T>`.
`ISet<T>`	This interface is implemented by sets. Sets allow combining different sets into a union, getting the intersection of two sets, and checking whether two sets overlap. `ISet<T>` derives from `ICollection<T>`.

INTERFACE	DESCRIPTION
`IDictionary<TKey, TValue>`	The interface `IDictionary<TKey, TValue>` is implemented by generic collection classes that have a key and a value. With this interface, all the keys and values can be accessed, items can be accessed with an indexer of type `TKey`, and items can be added or removed.
`ILookup<TKey, TValue>`	Similar to the `IDictionary<TKey, TValue>` interface, lookups have keys and values. However, with lookups, the collection can contain multiple values with one key.
`IComparer<T>`	The interface `IComparer<T>` is implemented by a comparer and used to sort elements inside a collection with the `Compare` method.
`IEqualityComparer<T>`	`IEqualityComparer<T>` is implemented by a comparer that can be used for keys in a dictionary. With this interface, the objects can be compared for equality.

LISTS

For resizable lists, .NET offers the generic class `List<T>`. This class implements the `IList`, `ICollection`, `IEnumerable`, `IList<T>`, `ICollection<T>`, and `IEnumerable<T>` interfaces.

The following examples use the members of the record `Racer` as elements to be added to the collection to represent a Formula 1 racer. This type has five properties: `Id`, `FirstName`, `LastName`, `Country`, and the number of `Wins` as specified with the positional record constructor. An overloaded constructor allows you to specify only four values when initializing the object. The method `ToString` is overridden to return the name of the racer. The record `Racer` also implements the generic interface `IComparable<T>` for sorting racer elements and `IFormattable` to allow passing custom format strings (code file `ListSamples/Racer.cs`):

```
public record Racer(int ID, string FirstName, string LastName, string Country,
  int Wins) : IComparable<Racer>, IFormattable
{
  public Racer(int id, string firstName, string lastName, string country)
    : this(id, firstName, lastName, country, Wins: 0)
  { }

  public override string ToString() => $"{FirstName} {LastName}";

  public string ToString(string? format, IFormatProvider? formatProvider) =>
    format?.ToUpper() switch
    {
      null => ToString(),
      "N" => ToString(),
      "F" => FirstName,
      "L" => LastName,
      "W" => $"{ToString()}, Wins: {Wins}",
      "C" => Country,
      "A" => $"{ToString()}, Country: {Country}, Wins: {Wins}",
```

```
        _ => throw new FormatException(string.Format(formatProvider,
          "Format {0} is not supported", format))
      };

    public string? ToString(string format) => ToString(format, null);

    public int CompareTo(Racer? other)
    {
      int compare = LastName?.CompareTo(other?.LastName) ?? -1;
      if (compare == 0)
      {
        return FirstName?.CompareTo(other?.FirstName) ?? -1;
      }
      return compare;
    }
  }
```

Creating Lists

You can create list objects by invoking the default constructor. With the generic class `List<T>`, you must specify the type for the values of the list with the declaration. The following code shows how to declare a `List<T>` with `int` and a list with `Racer` elements. `ArrayList` is a nongeneric list that accepts any `Object` type for its elements.

```
List<int> intList = new();
List<Racer> racers = new();
```

Using the default constructor creates an empty list. As soon as elements are added to the list, the capacity of the list is extended to allow 4 elements. If the fifth element is added, the list is resized to allow 8 elements. If 8 elements are not enough, the list is resized again to contain 16 elements. With every resize, the capacity of the list is doubled.

When the capacity of the list changes, the complete collection is reallocated to a new memory block. With the implementation of `List<T>`, an array of type `T` is used. With reallocation, a new array is created, and `Array.Copy` copies the elements from the old array to the new array. To save time, if you know the number of elements that should be in the list in advance, you can define the capacity with the constructor. The following example creates a collection with a capacity of 10 elements. If the capacity is not large enough for the elements added, the capacity is resized to 20 and then to 40 elements—doubled again:

```
List<int> intList = new(10);
```

You can get and set the capacity of a collection by using the `Capacity` property:

```
intList.Capacity = 20;
```

The capacity is not the same as the number of elements in the collection. The number of elements in the collection can be read with the `Count` property. Of course, the capacity is always larger or equal to the number of items. As long as no element was added to the list, the count is 0:

```
Console.WriteLine(intList.Count);
```

If you are finished adding elements to the list and don't want to add any more, you can get rid of the unneeded capacity by invoking the `TrimExcess` method; however, because the relocation takes time, `TrimExcess` has no effect if the item count is more than 90 percent of capacity:

```
intList.TrimExcess();
```

Collection Initializers

You can also assign values to collections using collection initializers. The syntax of collection initializers is similar to array initializers, which are explained in Chapter 6. With a collection initializer, values are assigned to the collection within curly brackets at the time the collection is initialized:

```
List<int> intList = new() {1, 2};
List<string> stringList = new() { "one", "two" };
```

> **NOTE** *Collection initializers are not reflected within the IL code of the compiled assembly. The compiler converts the collection initializer to invoke the* `Add` *method for every item from the initializer list.*

Adding Elements

You can add elements to the list with the `Add` method, shown in the following example. The generic instantiated type defines the parameter type of the `Add` method:

```
List<int> intList = new();
intList.Add(1);
intList.Add(2);
List<string> stringList = new();
stringList.Add("one");
stringList.Add("two");
```

The variable `racers` is defined as type `List<Racer>`. With the `new` operator, a new object of the same type is created. Because the class `List<T>` was instantiated with the concrete class `Racer`, now only `Racer` objects can be added with the `Add` method. In the following sample code, five Formula 1 racers are created and added to the collection. The first three are added using the collection initializer, and the last two are added by explicitly invoking the `Add` method (code file `ListSamples/Program.cs`):

```
Racer graham = new(7, "Graham", "Hill", "UK", 14);
Racer emerson = new(13, "Emerson", "Fittipaldi", "Brazil", 14);
Racer mario = new(16, "Mario", "Andretti", "USA", 12);
List<Racer> racers = new(20) {graham, emerson, mario};
racers.Add(new Racer(24, "Michael", "Schumacher", "Germany", 91));
racers.Add(new Racer(27, "Mika", "Hakkinen", "Finland", 20));
```

With the `AddRange` method of the `List<T>` class, you can add multiple elements to the collection at once. The method `AddRange` accepts an object of type `IEnumerable<T>`, so you can also pass an array as shown here (code file `ListSamples/Program.cs`):

```
racers.AddRange(new Racer[] {
  new(14, "Niki", "Lauda", "Austria", 25),
  new(21, "Alain", "Prost", "France", 51)});
```

> **NOTE** *The collection initializer can be used only during declaration of the collection. The* `AddRange` *method can be invoked after the collection is initialized. In case you get the data dynamically after creating the collection, you need to invoke* `AddRange`.

If you know some elements of the collection when instantiating the list, you can also pass any object that implements `IEnumerable<T>` to the constructor of the class. This is similar to the `AddRange` method (code file `ListSamples/Program.cs`):

```
List<Racer> racers = new(
  new Racer[] {
    new (12, "Jochen", "Rindt", "Austria", 6),
    new (22, "Ayrton", "Senna", "Brazil", 41) });
```

Inserting Elements

You can insert elements at a specified position with the `Insert` method (code file `ListSamples/Program.cs`):

```
racers.Insert(3, new Racer(6, "Phil", "Hill", "USA", 3));
```

The method `InsertRange` offers the capability to insert a number of elements, similar to the `AddRange` method shown earlier.

If the index set is larger than the number of elements in the collection, an exception of type `ArgumentOutOfRangeException` is thrown.

Accessing Elements

All classes that implement the `IList` and `IList<T>` interfaces offer an indexer, so you can access the elements by using an indexer and passing the item number. The first item can be accessed with an index value 0. By specifying `racers[3]`, for example, you access the fourth element of the list:

```
Racer r1 = racers[3];
```

When you use the `Count` property to get the number of elements, you can do a `for` loop to iterate through every item in the collection, and you can use the indexer to access every item (code file `ListSamples/Program.cs`):

```
for (int i = 0; i < racers.Count; i++)
{
  Console.WriteLine(racers[i]);
}
```

Because `List<T>` implements the interface `IEnumerable`, you can iterate through the items in the collection using the `foreach` statement as well (code file `ListSamples/Program.cs`):

```
foreach (var r in racers)
{
  Console.WriteLine(r);
}
```

> **NOTE** *Chapter 6 explains how the* `foreach` *statement is resolved by the compiler to make use of the* `IEnumerable` *and* `IEnumerator` *interfaces.*

Removing Elements

You can remove elements by index or by passing the item that should be removed. Here, the fourth element is removed from the collection:

```
racers.RemoveAt(3);
```

Instead of using the `RemoveAt` method, you can also directly pass a `Racer` object to the `Remove` method to remove this element. However, removing by index with the `RemoveAt` method is faster. The `Remove` method first searches in the collection to get the index of the item with the `IndexOf` method and then uses the index to remove the item. `IndexOf` first checks whether the item type implements the interface `IEquatable<T>`. If it does, the `Equals` method of this interface is invoked to find the item in the collection that is the same as the one passed to the method. If this interface is not implemented, the `Equals` method of the `Object` class is used to compare the items. The default implementation of the `Equals` method in the `Object` class does a bitwise comparison with value types, but compares only references with reference types.

> **NOTE** *Chapter 5, "Operators and Casts," explains how you can override the* `Equals` *method.*

In the following example, the racer referenced by the variable `graham` is removed from the collection (code file `ListSamples/Program.cs`):

```
if (!racers.Remove(graham))
{
  Console.WriteLine("object not found in collection");
}
```

The method `RemoveRange` removes a number of items from the collection. The first parameter specifies the index where the removal of items should begin; the second parameter specifies the number of items to be removed:

```
int index = 3;
int count = 5;
racers.RemoveRange(index, count);
```

To remove all items with some specific characteristics from the collection, you can use the `RemoveAll` method. This method uses the `Predicate<T>` parameter when searching for elements, which is discussed next. To remove all elements from the collection, use the `Clear` method defined with the `ICollection<T>` interface.

Searching

There are different ways to search for elements in the collection. You can get the index to the found item or a reference to the item itself. You can use methods such as `IndexOf`, `LastIndexOf`, `FindIndex`, `FindLastIndex`, `Find`, and `FindLast`. To just check whether an item exists, the `List<T>` class offers the `Exists` method.

The method `IndexOf` requires an object as a parameter and returns the index of the item if it is found inside the collection. If the item is not found, –1 is returned. Remember that `IndexOf` is using the `IEquatable<T>` interface to compare the elements (code file `ListSamples/Program.cs`):

```
int index1 = racers.IndexOf(mario);
```

With the `IndexOf` method, you can also specify that the complete collection should not be searched, and instead specify an index where the search should start and the number of elements that should be iterated for the comparison. To start from the end of the list to search for the index, you can use the `LastIndexOf` method.

Instead of searching a specific item with the `IndexOf` method, you can search for an item that has some specific characteristics that you can define with the `FindIndex` method. `FindIndex` requires a parameter of type `Predicate`:

```
public int FindIndex(Predicate<T> match);
```

The `Predicate<T>` type is a delegate that returns a Boolean value and requires type `T` as a parameter. If the predicate returns `true`, there's a match, and the element is found. If it returns `false`, the element is not found, and the search continues.

```
public delegate bool Predicate<T>(T obj);
```

With the `List<T>` class that is using `Racer` objects for type `T`, you can pass the address of a method that returns a `bool` and defines a parameter of type `Racer` to the `FindIndex` method. Finding the first racer of a specific country, you can use a lambda expression with a `Racer` parameter and a `bool` return as specified by the delegate. The following code uses a lambda expression that defines the implementation to search for an item where the `Country` property is set to `Finland`:

```
int index2 = racers.FindIndex(r => r.Country == "Finland");
```

Similar to the `IndexOf` method, with the `FindIndex` method you can also specify the index where the search should start and the count of items that should be iterated through. To do a search for an index beginning from the last element in the collection, you can use the `FindLastIndex` method.

The method `FindIndex` method returns the index of the found item. Instead of getting the index, you can also go directly to the item in the collection. The `Find` method requires a parameter of type `Predicate<T>`, much like the `FindIndex` method. The `Find` method in the following example searches for the first racer in the list who has the `FirstName` property set to `Niki`. Of course, you can also do a `FindLast` search to find the last item that fulfills the predicate.

```
Racer racer = racers.Find(r => r.FirstName == "Niki");
```

To get not only one but all the items that fulfill the requirements of a predicate, you can use the `FindAll` method. The `FindAll` method uses the same `Predicate<T>` delegate as the `Find` and `FindIndex` methods. The `FindAll` method does not stop when the first item is found; instead, the `FindAll` method iterates through every item in the collection and returns all items for which the predicate returns `true`.

With the `FindAll` method invoked in the next example, all `Racer` items are returned where the property `Wins` is set to more than 20. All racers who won more than 20 races are referenced from the `bigWinners` list:

```
List<Racer> bigWinners = racers.FindAll(r => r.Wins > 20);
```

Iterating through the variable `bigWinners` with a `foreach` statement gives the following result:

```
foreach (Racer r in bigWinners)
{
  Console.WriteLine($"{r:A}");
}
Michael Schumacher, Germany Wins: 91
Niki Lauda, Austria Wins: 25
Alain Prost, France Wins: 51
```

The result is not sorted, but the next section covers that.

Sorting

The `List<T>` class enables sorting its elements by using the `Sort` method. `Sort` uses the Quicksort algorithm whereby all elements are compared until the complete list is sorted. You can use several overloads of the `Sort` method. You can pass a delegate of type `Comparison<T>`, and an object implementing `IComparer<T>`. Using the `Sort` method without arguments is possible only if the elements in the collection implement the interface `IComparable`.

With the `IComparable<T>` implementation of the `Racer` type, the `Sort` method sorts racers by last name followed by first name:

```
racers.Sort();
```

If you need to do a sort other than the default supported by the item types, you need to use other techniques, such as passing an object that implements the `IComparer<T>` interface.

The class `RacerComparer` implements the interface `IComparer<T>` for `Racer` types. This class enables you to sort by the first name, last name, country, or number of wins. The kind of sort that should be done is defined with the inner enumeration type `CompareType`. The `CompareType` is set with the constructor of the class `RacerComparer`. The interface `IComparer<Racer>` defines the method `Compare`, which is required for sorting. In the implementation of this method, the `Compare` and `CompareTo` methods of the `string` and `int` types are used (code file `ListSamples/RacerComparer.cs`):

```
public class RacerComparer : IComparer<Racer>
{
  public enum CompareType
  {
    FirstName,
    LastName,
    Country,
    Wins
  }

  private CompareType _compareType;
  public RacerComparer(CompareType compareType) =>
    _compareType = compareType;

  public int Compare(Racer? x, Racer? y)
  {
    if (x is null && y is null) return 0;
    if (x is null) return -1;
    if (y is null) return 1;

    int CompareCountry(Racer x, Racer y)
    {
      int result = string.Compare(x.Country, y.Country);
      if (result == 0)
      {
        result = string.Compare(x.LastName, y.LastName);
      }
      return result;
    }

    return _compareType switch
    {
      CompareType.FirstName => string.Compare(x.FirstName, y.FirstName),
      CompareType.LastName => string.Compare(x.LastName, y.LastName),
      CompareType.Country => CompareCountry(x, y),
      CompareType.Wins => x.Wins.CompareTo(y.Wins),
      _ => throw new ArgumentException("Invalid Compare Type")
    };
  }
}
```

> **NOTE** *The* `Compare` *method returns 0 if the two elements passed to it are equal with the order. If a value less than 0 is returned, the first argument is less than the second. With a value larger than 0, the first argument is greater than the second. Passing* `null` *with an argument, the method shouldn't throw a* `NullReferenceException` *or* `ArgumentNullException`*. Instead,* `null` *should take its place before any other element; thus, –1 is returned if the first argument is* `null`*, and +1 if the second argument is* `null` *and 0 if both arguments are* `null`*.*

You can now use an instance of the `RacerComparer` class with the `Sort` method. Passing the enumeration `RacerComparer.CompareType.Country` sorts the collection by the property `Country`:

```
racers.Sort(new RacerComparer(RacerComparer.CompareType.Country));
```

Another way to do the sort is by using the overloaded `Sort` method, which requires a `Comparison<T>` delegate. `Comparison<T>` is a delegate to a method that has two parameters of type `T` and a return type `int`. If the parameter values are equal, the method must return 0. If the first parameter is less than the second, a value less than zero must be returned; otherwise, a value greater than zero is returned:

```
public delegate int Comparison<T>(T x, T y);
```

Now you can pass a lambda expression to the `Sort` method to do a sort by the number of wins. The two parameters are of type `Racer`, and in the implementation the `Wins` properties are compared by using the `int` method `CompareTo`. Also in the implementation, `r2` and `r1` are used in reverse order, so the number of wins is sorted in descending order. After the method has been invoked, the complete racer list is sorted based on the racer's number of wins:

```
racers.Sort((r1, r2) => r2.Wins.CompareTo(r1.Wins));
```

You can also reverse the order of a complete collection by invoking the `Reverse` method.

Read-Only Collections

After collections are created, they are read/write, of course; otherwise, you couldn't fill them with any values. However, after the collection is filled, you can create a read-only collection. The `List<T>` collection has the method `AsReadOnly` that returns an object of type `ReadOnlyCollection<T>`. The class `ReadOnlyCollection<T>` implements the same interfaces as `List<T>`, but all methods and properties that change the collection throw a `NotSupportedException`. Besides the interfaces of `List<T>`, `ReadOnlyCollection<T>` also implements the interfaces `IReadOnlyCollection<T>` and `IReadOnlyList<T>`. With the members of these interfaces, the collection cannot be changed.

Queues

A *queue* is a collection whose elements are processed according to first in, first out (FIFO) order, meaning the item that is put first in the queue is read first. Examples of queues are standing in line at the airport, a human resources queue to process employee applicants, print jobs waiting to be processed in a print queue, and a thread waiting for the CPU in a round-robin fashion. Sometimes the elements of a queue differ in their priority. For example, in the queue at the airport, business passengers are processed before economy passengers. In this case, multiple queues can be used, one queue for each priority. At the airport this is easily handled with separate check-in queues for business and economy passengers. The same is true for print queues and threads. You can have an array or a list of queues whereby one item in the array stands for a priority. Within every array item there's a queue where processing happens using the FIFO principle.

> **NOTE** *Later in this chapter, a different implementation with a linked list is used to define a list of priorities.*

A queue is implemented with the `Queue<T>` class. Internally, the `Queue<T>` class uses an array of type `T`, similar to the `List<T>` type. It implements the interfaces `IEnumerable<T>` and `ICollection`, but it doesn't implement `ICollection<T>` because this interface defines `Add` and `Remove` methods that shouldn't be available for queues.

The `Queue<T>` class does not implement the interface `IList<T>`, so you cannot access the queue using an indexer. The queue just allows you to add an item to the end of the queue (with the `Enqueue` method) and to get items from the head of the queue (with the `Dequeue` method).

Figure 8-1 shows the items of a queue. The `Enqueue` method adds items to one end of the queue; the items are read and removed at the other end of the queue with the `Dequeue` method. Invoking the `Dequeue` method once more removes the next item from the queue.

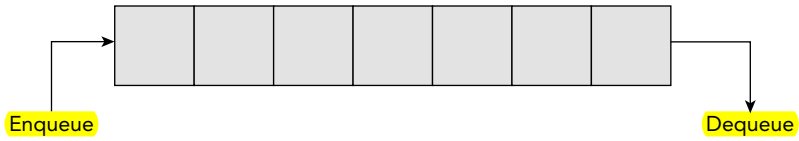

FIGURE 8-1

The following table describes the important methods of the `Queue<T>` class:

SELECTED QUEUE <T> MEMBERS	DESCRIPTION
`Count`	Returns the number of items in the queue.
`Enqueue`	Adds an item to the end of the queue.
`Dequeue`	Reads and removes an item from the head of the queue. If there are no more items in the queue when the `Dequeue` method is invoked, an exception of type `InvalidOperationException` is thrown.
`Peek`	Reads an item from the head of the queue but does not remove the item.
`TrimExcess`	Resizes the capacity of the queue. The `Dequeue` method removes items from the queue, but it doesn't resize the capacity of the queue. To get rid of the empty items at the beginning of the queue, use the `TrimExcess` method.

When creating queues, you can use constructors similar to those used with the `List<T>` type. The default constructor creates an empty queue, but you can also use a constructor to specify the capacity. With an overload of the constructor, you can also pass any other collection that implements the `IEnumerable<T>` interface that is copied to the queue.

The following example demonstrating the use of the `Queue<T>` class is a document management application. One task is used to add documents to the queue, and another task reads documents from the queue and processes them.

> **NOTE** *To make the queue sample more interesting, different tasks are used to work with the queue. One task writes messages to the queue, and another task reads messages from the queue. Reading and writing happens after a random time delay; you can monitor how the queue grows larger and smaller. Tasks are used in a simple way here, but you might want to read Chapter 11, "Tasks and Asynchronous Programming," before getting into this sample code.*

The items stored in the queue are of type `Document`. The record `Document` defines a title and content (code file `QueueSample/Document.cs`):

```
public record Document(string Title, string Content);
```

The `DocumentManager` class is a thin layer around the `Queue<T>` class. It defines how to handle documents: adding documents to the queue with the `AddDocument` method and getting documents from the queue with the `GetDocument` method.

Inside the `AddDocument` method, the document is added to the end of the queue using the `Enqueue` method. The first document from the queue is read with the `Dequeue` method inside `GetDocument`. Because multiple tasks can access the `DocumentManager` concurrently, access to the queue is locked with the `lock` statement. The `AddDocument` method returns the number of items in the queue to allow monitoring the queue size.

`IsDocumentAvailable` is a read-only Boolean property that returns `true` if there are documents in the queue and `false` if there are't (code file `QueueSample/DocumentManager.cs`):

```
public class DocumentManager
{
  private readonly object _syncQueue = new object();
  private readonly Queue<Document> _documentQueue = new();

  public int AddDocument(Document doc)
  {
    lock (_syncQueue)
    {
      _documentQueue.Enqueue(doc);
      return _documentQueue.Count;
    }
  }

  public Document GetDocument()
  {
    Document doc = null;
    lock (_syncQueue)
    {
      doc = _documentQueue.Dequeue();
    }
    return doc;
  }

  public bool IsDocumentAvailable => _documentQueue.Count > 0;
}
```

The class `ProcessDocuments` processes documents from the queue in a separate task. The only method that can be accessed from the outside is `Start`. In the `StartAsync` method, a new task is instantiated. A `ProcessDocuments` object is created to start the task, and the `RunAsync` method is defined as the start method of the task. With the `Task.Run` method, you can pass an `Action` delegate. Here, the `RunAsync` instance method of the `ProcessDocuments` class is invoked from the task.

With the `RunAsync` method of the `ProcessDocuments` class, a `do...while` loop is defined. Within this loop, the property `IsDocumentAvailable` is used to determine whether there is a document in the queue. If so, the document is taken from the `DocumentManager` and processed. If the task waits more than five seconds, waiting stops. Processing in this example is writing information only to the console. In a real application, the document could be written to a file, written to the database, or sent across the network (code file `QueueSample/ProcessDocuments.cs`):

```
public class ProcessDocuments
{
  public static Task StartAsync(DocumentManager dm) =>
    Task.Run(new ProcessDocuments(dm).RunAsync);

  protected ProcessDocuments(DocumentManager dm) =>
    _documentManager = dm ?? throw new ArgumentNullException(nameof(dm));

  private readonly DocumentManager _documentManager;

  protected async Task RunAsync()
  {
    Random random = new();
    Stopwatch stopwatch = new();
    stopwatch.Start();
    bool stop = false;
    do
    {
      if (stopwatch.Elapsed >= TimeSpan.FromSeconds(5))
      {
        stop = true;
      }
      if (_documentManager.IsDocumentAvailable)
      {
        stopwatch.Restart();
        Document doc = _documentManager.GetDocument();
        Console.WriteLine($"Processing document {doc.Title}");
      }
      // wait a random time before processing the next document
      await Task.Delay(random.Next(20));
    } while (!stop) ;
    Console.WriteLine("stopped reading documents");
  }
}
```

With the start of the application, a `DocumentManager` object is instantiated, and the document processing task is started. Then 1,000 documents are created and added to the `DocumentManager` (code file `QueueSample/Program.cs`):

```
DocumentManager dm = new();

Task processDocuments = ProcessDocuments.StartAsync(dm);
```

```
// Create documents and add them to the DocumentManager
Random random = new();
for (int i = 0; i < 1000; i++)
{
  var doc = new Document($"Doc {i}", "content");
  int queueSize = dm.AddDocument(doc);
  Console.WriteLine($"Added document {doc.Title}, queue size: {queueSize}");
  await Task.Delay(random.Next(20));
}
Console.WriteLine($"finished adding documents");
await processDocuments;
Console.WriteLine("bye!");
```

When you start the application, the documents are added to and removed from the queue, and you get output similar to the following:

```
Added document Doc 318, queue size: 6
Added document Doc 319, queue size: 7
Processing document Doc 313
Added document Doc 320, queue size: 7
Processing document Doc 314
Processing document Doc 315
Added document Doc 321, queue size: 7
Processing document Doc 316
```

A real-life scenario using the task described with the sample application might be an application that processes documents received with a Web API service.

STACKS

A stack is another container that is similar to the queue. You just use different methods to access the stack. The item that is added last to the stack is read first, so the stack is a last in, first out (LIFO) container.

Figure 8-2 shows the representation of a stack where the `Push` method adds an item to the stack, and the `Pop` method gets the item that was added last.

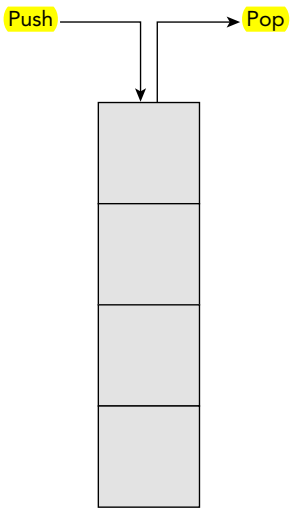

FIGURE 8-2

Similar to the `Queue<T>` class, the `Stack<T>` class implements the interfaces `IEnumerable<T>` and `ICollection`.

Important members of the `Stack<T>` class are listed in the following table:

SELECTED STACK<T> MEMBERS	DESCRIPTION
`Count`	Returns the number of items in the stack.
`Push`	Adds an item on top of the stack.
`Pop`	Removes and returns an item from the top of the stack. If the stack is empty, an exception of type `InvalidOperationException` is thrown.
`Peek`	Returns an item from the top of the stack but does not remove the item.
`Contains`	Checks whether an item is in the stack and returns `true` if it is.

In this example, three items are added to the stack with the `Push` method. With the `foreach` method, all items are iterated using the `IEnumerable` interface. The enumerator of the stack does not remove the items; it just returns them item by item (code file `StackSample/Program.cs`):

```
Stack<char> alphabet = new();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');
foreach (char item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();
```

Because the items are read in order from the last item added to the first, the following result is produced:

```
CBA
```

Reading the items with the enumerator does not change the state of the items. With the `Pop` method, every item that is read is also removed from the stack. This way, you can iterate the collection using a `while` loop and verify the `Count` property if items still exist:

```
Stack<char> alphabet = new();
alphabet.Push('A');
alphabet.Push('B');
alphabet.Push('C');
Console.Write("First iteration: ");
foreach (char item in alphabet)
{
  Console.Write(item);
}
Console.WriteLine();
Console.Write("Second iteration: ");
while (alphabet.Count > 0)
{
  Console.Write(alphabet.Pop());
}
Console.WriteLine();
```

The result gives CBA twice—once for each iteration. After the second iteration, the stack is empty because the second iteration used the `Pop` method:

```
First iteration: CBA
Second iteration: CBA
```

LINKED LISTS

`LinkedList<T>` is a doubly linked list, whereby one element references the next and the previous one, as shown in Figure 8-3. This way you can easily walk forward through the complete list by moving to the next element or backward by moving to the previous element.

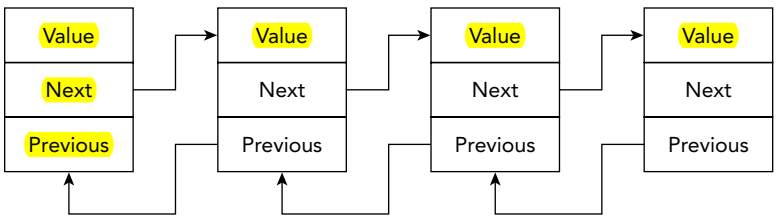

FIGURE 8-3

The advantage of a linked list is that if items are inserted anywhere in the list, the linked list is very fast. When an item is inserted, only the `Next` reference of the previous item and the `Previous` reference of the next item must be changed to reference the inserted item. With the `List<T>` class, when an element is inserted, all subsequent elements must be moved.

Of course, there's also a disadvantage with linked lists. Items of linked lists can be accessed only one after the other. It takes a long time to find an item that's somewhere in the middle or at the end of the list.

A linked list cannot just store the items inside the list; together with every item, the linked list must have information about the next and previous items. That's why the `LinkedList<T>` contains items of type `LinkedListNode<T>`. With the class `LinkedListNode<T>`, you can get to the next and previous items in the list. The `LinkedListNode<T>` class defines the properties `List`, `Next`, `Previous`, and `Value`. The `List` property returns the `LinkedList<T>` object that is associated with the node. `Next` and `Previous` are for iterating through the list and accessing the next or previous item. `Value` returns the item that is associated with the node. `Value` is of type `T`.

The `LinkedList<T>` class itself defines members to access the first (`First`) and last (`Last`) items of the list, to insert items at specific positions (`AddAfter`, `AddBefore`, `AddFirst`, `AddLast`), to remove items from specific positions (`Remove`, `RemoveFirst`, `RemoveLast`), and to find elements where the search starts from either the beginning (`Find`) or the end (`FindLast`) of the list.

With the sample application, a `Document` record is defined to put it into the LinkedList (code file `LinkedListSample/Program.cs`):

```
record Document(int Id, string Text);
```

The following code snippet creates a `LinkedList` and adds the first item to the start of the list using `AddFirst`. The `AddFirst` method returns a `LinkedListNode` object that's used with the next invocation to the list to add the document with an ID of 2 after the first object. The document with an ID of 3 is added last to the list using the `AddLast` method (which is also at that stage after the document with ID of 2). Using the `AddBefore` method, the document with an ID of 4 is added before the last one. After filling the list, the `LinkedList` is iterated using the `foreach` statement:

```
LinkedList<Document> list = new();
LinkedListNode<Document> first = list.AddFirst(new Document(1, "first"));
```

```
list.AddAfter(first, new Document(2, "after first"));
LinkedListNode<Document> last = list.AddLast(new Document(3, "Last"));
Document doc4 = new(4, "before last");
list.AddBefore(last, doc4);

foreach (var item in list)
{
  Console.WriteLine(item);
}
```

Instead of using the `foreach` statement, you can easily iterate through all elements of the collection by accessing the `Next` property of every `LinkedListNode`:

```
void IterateUsingNext(LinkedListNode<Document> start)
{
  if (start.Value is null) return;
  LinkedListNode<Document>? current = start;
  do
  {
    Console.WriteLine(current.Value);
    current = current.Next;
  } while (current is not null);
}
```

The method `IterateUsingNext` is invoked from the top-level statements passing the first object:

```
if (list.First is not null)
{
    IterateUsingNext(list.First);
}
```

Running the application, you'll see two times the documents iterated. One iteration is shown here:

```
Document { Id = 1, Text = first }
Document { Id = 2, Text = after first }
Document { Id = 4, Text = before last }
Document { Id = 3, Text = Last }
```

Using the `Remove` method passing a `Document` object requires the `Remove` method to iterate through the collection until the `Document` can be found and removed:

```
list.Remove(doc4);

Console.WriteLine("after removal");
foreach (var item in list)
{
    Console.WriteLine(item);
}
```

Later in this chapter, in the section "Performance," you'll see a table with the big-O notation where you can compare the performance of different collection classes based on the operations, so you can decide for the collection type to use more easily.

SORTED LIST

If the collection you need should be sorted based on a key, you can use `SortedList<TKey, TValue>`. This class sorts the elements based on a key. You can use any type for the value and also for the key.

The following example creates a sorted list for which both the key and the value are of type `string`. The default constructor creates an empty list, and then two books are added with the `Add` method. With overloaded constructors, you can define the capacity of the list and pass an object that implements the interface `IComparer<TKey>`, which is used to sort the elements in the list.

The first parameter of the `Add` method is the key (the book title); the second parameter is the value (the ISBN). Instead of using the `Add` method, you can use the indexer to add elements to the list. The indexer requires the key as index parameter. If a key already exists, the `Add` method throws an exception of type `ArgumentException`. If the same key is used with the indexer, the new value replaces the old value (code file `SortedListSample/Program.cs`):

```
SortedList<string, string> books = new();
books.Add("Front-end Development with ASP.NET Core", "978-1-119-18140-8");
books.Add("Beginning C# 7 Programming", "978-1-119-45866-1");

books["Enterprise Services"] = "978-0321246738";
books["Professional C# 7 and .NET Core 2.1"] = "978-1-119-44926-3";
```

> **NOTE** `SortedList<TKey, TValue>` *allows only one value per key. If you need multiple values per key, you can use* `Lookup<TKey, TElement>`.

You can iterate through the list using a `foreach` statement. Elements returned by the enumerator are of type `KeyValuePair<TKey, TValue>`, which contains both the key and the value. The key can be accessed with the `Key` property, and the value can be accessed with the `Value` property:

```
foreach (KeyValuePair<string, string> book in books)
{
  Console.WriteLine($"{book.Key}, {book.Value}");
}
```

The iteration displays book titles and ISBN numbers ordered by the key:

```
Beginning C# 7 Programming, 978-1-119-45866-1
Enterprise Services, 978-0321246738
Front-end Development with ASP.NET Core, 978-1-119-18140-8
Professional C# 7 and .NET Core 2.1, 978-1-119-44926-3
```

You can also access the values and keys by using the `Values` and `Keys` properties. The `Values` property returns `IList<TValue>`, and the `Keys` property returns `IList<TKey>`, so you can use these properties with a `foreach`:

```
foreach (string isbn in books.Values)
{
  Console.WriteLine(isbn);
}
foreach (string title in books.Keys)
{
  Console.WriteLine(title);
}
```

The first loop displays the values and next the keys:

```
978-1-119-45866-1
978-0321246738
978-1-119-18140-8
978-1-119-44926-3
Beginning C# 7 Programming
```

```
Enterprise Services
Front-end Development with ASP.NET Core
Professional C# 7 and .NET Core 2.1
```

If you try to access an element with an indexer and pass a key that does not exist, an exception of type `KeyNotFoundException` is thrown. To avoid that exception, you can use the method `ContainsKey`, which returns `true` if the key passed exists in the collection, or you can invoke the method `TryGetValue`, which tries to get the value but doesn't throw an exception if it isn't found:

```
string title = "Professional C# 10";
if (!books.TryGetValue(title, out string isbn))
{
  Console.WriteLine($"{title} not found");
}
else
{
  Console.WriteLine($"{title} found: {isbn}");
}
```

DICTIONARIES

A dictionary represents a sophisticated data structure that enables you to access an element based on a key. Dictionaries are also known as *hash tables* or *maps*. The main feature of dictionaries is fast lookup based on keys. You can also add and remove items freely, as with `List<T>`, but without the performance overhead of having to shift subsequent items in memory.

Figure 8-4 shows a simplified representation of a dictionary. Here employee IDs such as `B4711` are the keys added to the dictionary. The key is transformed into a hash. With the hash, a number is created to associate an index with the values. The index then contains a link to the value. The figure is simplified because it is possible for a single index entry to be associated with multiple values, and the index can be stored in a hash table.

.NET offers several dictionary classes. The main class you use is `Dictionary<TKey, TValue>`.

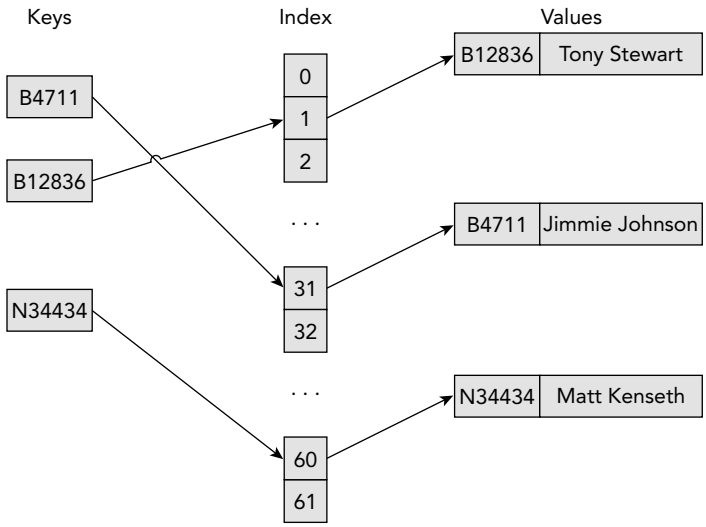

FIGURE 8-4

Dictionary Initializers

C# offers a syntax to initialize dictionaries at declaration with *dictionary initializers*. A dictionary with a key of `int` and a value of `string` can be initialized as follows:

```
Dictionary<int, string> dict = new()
{
  [3] = "three",
  [7] = "seven"
};
```

Here, two elements are added to the dictionary. The first element has a key of `3` and a string value `three`; the second element has a key of `7` and a string value `seven`. This initializer syntax is easily readable and is an adaptation of the collection initializer syntax shown earlier in this chapter.

Key Type

A type that is used as a key in the dictionary must override the method `GetHashCode` of the `Object` class. Whenever a dictionary class needs to determine where an item should be located, it calls the `GetHashCode` method. The `int` that is returned by `GetHashCode` is used by the dictionary to calculate an index of where to place the element. I won't go into this part of the algorithm; what you should know is that it involves prime numbers, so the capacity of a dictionary is a prime number.

The implementation of `GetHashCode` must satisfy the following requirements:

- The same object should always return the same value.
- Different objects can return the same value.
- It must not throw exceptions.
- It should use at least one instance field.
- The hash code should not change during the lifetime of the object.

Besides the requirements that must be satisfied by the `GetHashCode` implementation, it's also good practice to satisfy these requirements:

- It should execute as quickly as possible; it must be inexpensive to compute.
- The hash code value should be evenly distributed across the entire range of numbers that an `int` can store.

> **NOTE** *Good performance of the dictionary is based on a good implementation of the method* `GetHashCode`.

What's the reason for having hash code values evenly distributed across the range of integers? If two keys return hashes that have the same index, the dictionary class needs to start looking for the nearest available free location to store the second item—and it will have to do some searching to retrieve this item later. This is obviously going to hurt performance. In addition, if a lot of your keys tend to provide the same storage indexes for where they should be stored, this kind of clash becomes more likely. However, because of the way that Microsoft's part of the algorithm works, this risk is minimized when the calculated hash values are evenly distributed between `int.MinValue` and `int.MaxValue`.

Besides having an implementation of `GetHashCode`, the key type also must implement the `IEquatable<T>.Equals` method or override the `Equals` method from the `Object` class. Because different key objects may return

the same hash code, the method `Equals` is used by the dictionary comparing keys. The dictionary examines whether two keys, such as `A` and `B`, are equal; it invokes `A.Equals(B)`. This means you must ensure that the following is always true: `If A.Equals(B)` is true, then `A.GetHashCode` and `B.GetHashCode` must always return the same hash code.

This may seem a fairly subtle point, but it is crucial. If you contrived some way of overriding these methods so that the preceding statement were not always true, a dictionary that uses instances of this class as its keys would not work properly. Instead, you'd find funny things happening. For example, you might place an object in the dictionary and then discover that you could never retrieve it, or you might try to retrieve an entry and have the wrong entry returned.

> **NOTE** *For this reason, the C# compiler displays a compilation warning if you supply an override for* `Equals` *but don't supply an override for* `GetHashCode`.

For `System.Object`, this condition is true because `Equals` simply compares references, and `GetHashCode` actually returns a hash that is based solely on the address of the object. This means that hash tables based on a key that doesn't override these methods will work correctly. However, the problem with this approach is that keys are regarded as equal only if they are the same object. That means when you place an object in the dictionary, you have to hang onto the reference to the key; you can't simply instantiate another key object later with the same value. If you don't override `Equals` and `GetHashCode`, the type is not very convenient to use in a dictionary as a key.

Incidentally, `System.String` implements the interface `IEquatable` and overloads `GetHashCode` appropriately. `Equals` provides value comparison, and `GetHashCode` returns a hash based on the value of the string. Strings can be used conveniently as keys in dictionaries.

Number types such as `Int32` also implement the interface `IEquatable` and overload `GetHashCode`. However, the hash code returned by these types simply maps to the value. If the number you would like to use as a key is not itself distributed around the possible values of an integer, using integers as keys doesn't fulfill the rule of evenly distributing key values to get the best performance. `Int32` is not meant to be used in a dictionary.

A C# 9 record is a class, but offers value semantics. With this, it also implements the `IEquatable` interface and overrides `GetHashCode`.

If you need to use a key type that does not implement `IEquatable` and does not override `GetHashCode` according to the key values you store in the dictionary, you can create a comparer implementing the interface `IEqualityComparer<T>`. This interface defines the methods `GetHashCode` and `Equals` with an argument of the object passed, so you can offer an implementation different from the object type itself. An overload of the `Dictionary<TKey, TValue>` constructor allows passing an object implementing `IEqualityComparer<T>`. If such an object is assigned to the dictionary, this class is used to generate the hash codes and compare the keys.

Dictionary Example

The dictionary example in this section is a program that sets up a dictionary of employees. The dictionary is indexed by `EmployeeId` objects, and each item stored in the dictionary is an `Employee` object that stores details of an employee.

The struct `EmployeeId` is implemented to define a key to be used in a dictionary. The members of the class are a prefix character and a number for the employee. Both of these variables are read-only and can be initialized only in the constructor to ensure that keys within the dictionary cannot change. The default implementation of the record's `GetHashCode` uses all of its fields to generate the hash code. When you have read-only variables, it is guaranteed that they can't be changed. The fields are filled within the constructor. The `ToString` method is overloaded to get a string representation of the employee ID. As required for a key type, `EmployeeId`

implements the interface `IEquatable` and overloads the method `GetHashCode` (code file `DictionarySample/EmployeeId.cs`):

```
public class EmployeeIdException : Exception
{
  public EmployeeIdException(string message) : base(message) { }
}

public struct EmployeeId : IEquatable<EmployeeId>
{
  private readonly char _prefix;
  private readonly int _number;
  public EmployeeId(string id)
  {
    if (id == null) throw new ArgumentNullException(nameof(id));
    _prefix = (id.ToUpper())[0];
    int last = id.Length > 7 ? 7 : id.Length;
    try
    {
      _number = int.Parse(id[1..last]);
    }
    catch (FormatException)
    {
      throw new EmployeeIdException("Invalid EmployeeId format");
    }
  }

  public override string ToString() => _prefix.ToString() +
    $"{_number,6:000000}";

  public override int GetHashCode() => (_number ^ _number << 16) * 0x15051505;

  public bool Equals(EmployeeId other) =>
    _prefix == other._prefix && _number == other._number;

  public override bool Equals(object obj) => Equals((EmployeeId)obj);

  public static bool operator ==(EmployeeId left, EmployeeId right) =>
    left.Equals(right);

  public static bool operator !=(EmployeeId left, EmployeeId right) =>
    !(left == right);
}
```

The `Equals` method that is defined by the `IEquatable<T>` interface compares the values of two `EmployeeId` objects and returns `true` if both values are the same. Instead of implementing the `Equals` method from the `IEquatable<T>` interface, you can also override the `Equals` method from the `Object` class:

```
public bool Equals(EmployeeId other) =>
  _prefix == other._prefix && _number == other._number;
```

With the `_number` variable, a value from 1 to around 190,000 is expected for the employees. This doesn't fill the range of an integer. The algorithm used by `GetHashCode` shifts the number 16 bits to the left, then does an XOR (exclusive OR) with the original number, and finally multiplies the result by the hex value 15051505. The hash code is fairly evenly distributed across the range of an integer:

```
public override int GetHashCode() => (number ^ number << 16) * 0x1505_1505;
```

The `Employee` type is a simple record with private fields for the name, salary, and ID of the employee. The constructor initializes all values, and the method `ToString` returns a string representation of an instance. The implementation of `ToString` uses a format string to create the string representation for performance reasons (code file `DictionarySample/Employee.cs`):

```
public record Employee
{
  private readonly string _name;
  private readonly decimal _salary;
  private readonly EmployeeId _id;
  public Employee(EmployeeId id, string name, decimal salary)
  {
    _id = id;
    _name = name;
    _salary = salary;
  }

  public override string ToString() =>
    $"{_id.ToString()}: {_name, -20} {_salary,12:C}";
}
```

In the `Program.cs` file, a new `Dictionary<TKey, TValue>` instance is created, where the key is of type `EmployeeId` and the value is of type `Employee`. The constructor allocates a capacity of 31 elements. Remember that capacity is based on prime numbers. However, when you assign a value that is not a prime number, you don't need to worry. The `Dictionary<TKey, TValue>` class itself takes the next prime number from a list of specially selected prime numbers that follows the integer passed to the constructor to allocate the capacity. After creating the employee objects and IDs, they are added to the newly created dictionary using the new dictionary initializer syntax. Of course, you can also invoke the `Add` method of the dictionary to add objects instead (code file `DictionarySample/Program.cs`):

```
EmployeeId idKyle = new("J18");
Employee kyle = new Employee(idKyle, "Kyle Bush", 138_000.00m );

EmployeeId idMartin = new("J19");
Employee martin = new(idMartin, "Martin Truex Jr", 73_000.00m);

EmployeeId idKevin = new("S4");
Employee kevin = new(idKevin, "Kevin Harvick", 116_000.00m);

EmployeeId idDenny = new EmployeeId("J11");
Employee denny = new Employee(idDenny, "Denny Hamlin", 127_000.00m);

EmployeeId idJoey = new("T22");
Employee joey = new(idJoey, "Joey Logano", 96_000.00m);

EmployeeId idKyleL = new ("C42");
Employee kyleL = new (idKyleL, "Kyle Larson", 80_000.00m);


Dictionary<EmployeeId, Employee> employees = new(31)
{
  [idKyle] = kyle,
  [idMartin] = martin,
  [idKevin] = kevin,
```

```
  [idDenny] = denny,
  [idJoey] = joey,
};

foreach (var employee in employees.Values)
{
  Console.WriteLine(employee);
}
//...
```

After the entries are added to the dictionary, employees are read from the dictionary inside a `while` loop. The user is asked to enter an employee number to store in the variable `userInput`, and the user can exit the application by pressing the key `X`. If the key is in the dictionary, it is examined with the `TryGetValue` method of the `Dictionary<TKey, TValue>` class. `TryGetValue` returns `true` if the key is found or `false` otherwise. If the value is found, the value associated with the key is stored in the employee variable. This value is written to the console.

> **NOTE** *You can also use an indexer of the* `Dictionary<TKey, TValue>` *class instead of* `TryGetValue` *to access a value stored in the dictionary. However, if the key is not found, the indexer throws an exception of type* `KeyNotFoundException`.

```
while (true)
{
  Console.Write("Enter employee id (X to exit)> ");
  string? userInput = Console.ReadLine();
  userInput = userInput?.ToUpper();
  if (userInput == null || userInput == "X") break;

  try
  {
    EmployeeId id = new(userInput);
    if (!employees.TryGetValue(id, out Employee? employee))
    {
      Console.WriteLine($"Employee with id {id} does not exist");
    }
    else
    {
      Console.WriteLine(employee);
    }
  }
  catch (EmployeeIdException ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Running the application produces the following output:

```
J000018: Kyle Bush            $138.000,00
J000019: Martin Truex Jr       $73.000,00
S000004: Kevin Harvick        $116.000,00
```

```
J000011: Denny Hamlin         $127.000,00
T000022: Joey Logano           $96.000,00
Enter employee id (X to exit)> T22
T000022: Joey Logano           $96.000,00
Enter employee id (X to exit)> J18
J000018: Kyle Bush            $138.000,00
Enter employee id (X to exit)> X
```

Lookups

`Dictionary<TKey, TValue>` supports only one value per key. The class `Lookup<TKey, TElement>` resembles a `Dictionary<TKey, TValue>` but maps keys to a collection of values. This class is defined with the namespace `System.Linq`.

`Lookup<TKey, TElement>` cannot be created like the dictionary. Instead, you have to invoke the method `ToLookup`, which returns a `Lookup<TKey, TElement>` object. The method `ToLookup` is an extension method that is available with every class implementing `IEnumerable<T>`. In the following example, a list of `Racer` objects is filled. Because `List<T>` implements `IEnumerable<T>`, the `ToLookup` method can be invoked on the racers list. This method requires a delegate of type `Func<TSource, TKey>` that defines the selector of the key. Here, the racers are selected based on their country by using the lambda expression `r => r.Country`. The `foreach` loop accesses only the racers from Australia by using the indexer (code file `LookupSample/Program.cs`):

```
List<Racer> racers = new();
racers.Add(new Racer(26, "Jacques", "Villeneuve", "Canada", 11));
racers.Add(new Racer(18, "Alan", "Jones", "Australia", 12));
racers.Add(new Racer(11, "Jackie", "Stewart", "United Kingdom", 27));
racers.Add(new Racer(15, "James", "Hunt", "United Kingdom", 10));
racers.Add(new Racer(5, "Jack", "Brabham", "Australia", 14));

var lookupRacers = racers.ToLookup(r => r.Country);

foreach (Racer r in lookupRacers["Australia"])
{
  Console.WriteLine(r);
}
```

> **NOTE** *You can read more about extension methods in Chapter 9, "Language Integrated Query." Lambda expressions are explained in Chapter 7, "Delegates, Lambdas, and Events."*

The output shows the racers from Australia:

```
Alan Jones
Jack Brabham
```

Sorted Dictionaries

`SortedDictionary<TKey, TValue>` is a binary search tree in which the items are sorted based on the key. The key type must implement the interface `IComparable<TKey>`. If the key type is not sortable, you can also create a comparer implementing `IComparer<TKey>` and assign the comparer as a constructor argument of the sorted dictionary.

Earlier in this chapter you read about `SortedList<TKey, TValue>`. `SortedDictionary<TKey, TValue>` and `SortedList<TKey, TValue>` have similar functionality, but because `SortedList<TKey, TValue>` is implemented as a list that is based on an array, and `SortedDictionary<TKey, TValue>` is implemented as a tree, the classes have different characteristics:

- `SortedList<TKey, TValue>` uses less memory than `SortedDictionary<TKey, TValue>`.
- `SortedDictionary<TKey, TValue>` has faster insertion and removal of elements.
- When populating the collection with already sorted data, `SortedList<TKey, TValue>` is faster if capacity changes are not needed.

> **NOTE** `SortedList` *consumes less memory than* `SortedDictionary`. `SortedDictionary` *is faster with inserts and the removal of unsorted data.*

SETS

A collection that contains only distinct items is known by the term *set*. .NET Core includes two sets, `HashSet<T>` and `SortedSet<T>`, that both implement the interface `ISet<T>`. `HashSet<T>` contains a hash table of distinct items that is unordered; with `SortedSet<T>`, the list is ordered.

The `ISet<T>` interface offers methods to create a union of multiple sets, to create an intersection of sets, or to provide information if one set is a superset or subset of another.

In the following sample code, three new sets of type `string` are created and filled with Formula 1 cars. The `HashSet<T>` class implements the `ICollection<T>` interface. However, the `Add` method is implemented explicitly, and a different `Add` method is offered by the class, as you can see in the following code snippet. The `Add` method differs by the return type; a Boolean value is returned to provide the information if the element was added. If the element was already in the set, it is not added, and `false` is returned (code file `SetSample/Program.cs`):

```
HashSet<string> companyTeams = new()
{ "Ferrari", "McLaren", "Mercedes" };

HashSet<string> traditionalTeams = new() { "Ferrari", "McLaren" };

HashSet<string> privateTeams = new()
{ "Red Bull", "Toro Rosso", "Force India", "Sauber" };

if (privateTeams.Add("Williams"))
{
  Console.WriteLine("Williams added");
}

if (!companyTeams.Add("McLaren"))
{
  Console.WriteLine("McLaren was already in this set");
}
```

The result of these two `Add` methods is written to the console:

```
Williams added
McLaren was already in this set
```

The methods `IsSubsetOf` and `IsSupersetOf` compare a set with a collection that implements the `IEnumerable<T>` interface and returns a Boolean result. Here, `IsSubsetOf` verifies whether every element in `traditionalTeams` is contained in `companyTeams`, which is the case; `IsSupersetOf` verifies whether `traditionalTeams` has any additional elements compared to `companyTeams`:

```
if (traditionalTeams.IsSubsetOf(companyTeams))
{
  Console.WriteLine("traditionalTeams is subset of companyTeams");
}
if (companyTeams.IsSupersetOf(traditionalTeams))
{
  Console.WriteLine("companyTeams is a superset of traditionalTeams");
}
```

The output of this verification is shown here:

```
traditionalTeams is a subset of companyTeams
companyTeams is a superset of traditionalTeams
```

Williams is a traditional team as well, which is why this team is added to the `traditionalTeams` collection:

```
traditionalTeams.Add("Williams");
if (privateTeams.Overlaps(traditionalTeams))
{
  Console.WriteLine("At least one team is the same with traditional " +
    "and private teams");
}
```

Because there's an overlap, this is the result:

```
At least one team is the same with traditional and private teams.
```

The variable `allTeams` that references a new `SortedSet<string>` is filled with a union of `companyTeams`, `privateTeams`, and `traditionalTeams` by calling the `UnionWith` method:

```
SortedSet<string> allTeams = new(companyTeams);
allTeams.UnionWith(privateTeams);
allTeams.UnionWith(traditionalTeams);
Console.WriteLine();
Console.WriteLine("all teams");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

Here, all teams are returned, but every team is listed just once because the set contains only unique values; and because the container is a `SortedSet<string>`, the result is ordered:

```
Ferrari
Force India
Lotus
McLaren
Mercedes
Red Bull
Sauber
Toro Rosso
Williams
```

The method `ExceptWith` removes all private teams from the `allTeams` set:

```
allTeams.ExceptWith(privateTeams);
Console.WriteLine();
Console.WriteLine("no private team left");
foreach (var team in allTeams)
{
  Console.WriteLine(team);
}
```

The remaining elements in the collection do not contain any private teams:

```
Ferrari
McLaren
Mercedes
```

PERFORMANCE

Many collection classes offer the same functionality as others; for example, `SortedList` offers nearly the same features as `SortedDictionary`. However, often there's a big difference in performance. Whereas one collection consumes less memory, the other collection class is faster with the retrieval of elements. The Microsoft documentation often provides performance hints about methods of the collection, giving you information about the time the operation requires in big-O notation:

- **O(1):** This means that the time this operation needs is constant no matter how many items are in the collection. For example, the `ArrayList` has an `Add` method with O(1) behavior. No matter how many elements are in the list, it always takes the same amount of time when adding a new element to the end of the list. The `Count` property provides the number of items, so it is easy to find the end of the list.
- **O(log n):** This means that the time needed for the operation increases with every element in the collection, but the increase of time for each element is not linear but logarithmic. `SortedDictionary<TKey, TValue>` has O(log n) behavior for inserting operations inside the collection; `SortedList<TKey, TValue>` has O(n) behavior for the same functionality. Here, `SortedDictionary<TKey, TValue>` is a lot faster because it is more efficient to insert elements into a tree structure than into a list.
- **O(n):** This means it takes the worst-case amount of time of n to perform an operation on the collection. The `Add` method of `ArrayList` can be an O(n) operation if a reallocation of the collection is required. Changing the capacity causes the list to be copied, and the time for the copy increases linearly with every element.

The following table lists collection classes and their performance for different actions such as adding, inserting, and removing items. With this table, you can select the best collection class for your purpose. The left column lists the collection class. The Add column gives timing information about adding items to the collection. The `List<T>` and the `HashSet<T>` classes define `Add` methods to add items to the collection. With other collection classes, use a different method to add elements to the collection; for example, the `Stack<T>` class defines a `Push` method, and the `Queue<T>` class defines an `Enqueue` method. You can find this information in the table as well.

If there are multiple big-O values in a cell, the reason is that if a collection needs to be resized, resizing takes a while. For example, with the `List<T>` class, adding items needs O(1). If the capacity of the collection is not large enough and the collection needs to be resized, the resize requires O(n) time. The larger the collection, the longer the resize operation takes. It's best to avoid resizes by setting the capacity of the collection when you create it to a value that can hold all the elements.

If the table cell contents is n/a, the operation is not applicable with this collection type.

COLLECTION	ADD	INSERT	REMOVE	ITEM	SORT	FIND
`List<T>`	O(1) or O(n) if the collection must be resized	O(n)	O(n)	O(1)	O (n log n), worst case O(n ^ 2)	O(n)
`Stack<T>`	`Push`, O(1), or O(n) if the stack must be resized	n/a	`Pop`, O(1)	n/a	n/a	n/a
`Queue<T>`	`Enqueue`, O(1), or O(n) if the queue must be resized	n/a	`Dequeue`, O(1)	n/a	n/a	n/a
`HashSet<T>`	O(1) or O(n) if the set must be resized	`Add` O(1) or O(n)	O(1)	n/a	n/a	n/a
`SortedSet<T>`	O(1) or O(n) if the set must be resized	Add O(1) or O(n)	O(1)	n/a	n/a	n/a
`LinkedList<T>`	`AddLast` O(1)	`Add After` O(1)	O(1)	n/a	n/a	O(n)
`Dictionary <TKey, TValue>`	O(1) or O(n)	n/a	O(1)	O(1)	n/a	n/a
`SortedDictionary<TKey, TValue>`	O(log n)	n/a	O(log n)	O(log n)	n/a	n/a
`SortedList <TKey, TValue>`	O(n) for unsorted data, O(log n) for end of list, O(n) if resize is needed	n/a	O(n)	O(log n) to read/ write, O(log n) if the key is in the list, O(n) if the key is not in the list	n/a	n/a

IMMUTABLE COLLECTIONS

If an object can change its state, it is hard to use it from multiple simultaneously running tasks. Synchronization is necessary with these collections. If an object cannot change state, it's a lot easier to use it from multiple threads. An object that can't change is an immutable object. Collections that cannot be changed are immutable collections.

> **NOTE** *The topics of using multiple tasks and threads and programming with asynchronous methods are explained in detail in Chapter 11, "Tasks and Asynchronous Programming," and Chapter 17, "Parallel Programming."*

When you compare read-only collections like those discussed earlier in this chapter with immutable collections, there's a big difference: read-only collections make use of an interface to mutable collections. Using this interface, the collection cannot be changed. However, if someone still has a reference to the mutable collection, it still can be changed. With immutable collections, nobody can change this collection.

Let's start with a simple immutable string array using the class `ImmutableArray`. This class is defined in the `System.Collections.Immutable` namespace. You can create the array with the static `Create` method as shown. The `Create` method is overloaded where other variants of this method allow passing any number of elements. Notice that two different types are used here: the nongeneric `ImmutableArray` class with the static `Create` method and the generic `ImmutableArray` struct that is returned from the `Create` method. In the following code snippet, an empty array is created (code file `ImmutableCollectionSample/Program.cs`):

```
ImmutableArray<string> a1 = ImmutableArray.Create<string>();
```

An empty array is not very useful. The `ImmutableArray<T>` type offers an `Add` method to add elements. However, contrary to other collection classes, the `Add` method does not change the immutable collection itself. Instead, a new immutable collection is returned. So, after the call of the `Add` method, `a1` is still an empty collection, and `a2` is an immutable collection with one element. The `Add` method returns the new immutable collection:

```
ImmutableArray<string> a2 = a1.Add("Williams");
```

With this, it is possible to use this API in a fluent way and invoke one `Add` method after the other. The variable `a3` now references an immutable collection containing four elements:

```
ImmutableArray<string> a3 =
  a2.Add("Ferrari").Add("Mercedes").Add("Red Bull Racing");
```

With each of these stages using the immutable array, the complete collections are not copied with every step. Instead, the immutable types make use of a shared state and copy the collection only when it's necessary.

However, it's even more efficient to first fill the collection and then make it an immutable array. When some manipulation needs to take place, you can again use a mutable collection. A builder class offered by the immutable types helps with that.

To see this in action, first an `Account` record is created that is put into the collection. This type itself is immutable and cannot be changed by using read-only auto properties (code file `ImmutableCollectionSample/Account.cs`):

```
public record Account(string Name, decimal Amount);
```

Next a `List<Account>` collection is created and filled with sample accounts (code file `ImmutableCollectionSample/Program.cs`):

```
List<Account> accounts = new()
{
  new("Scrooge McDuck", 667377678765m),
```

```
    new("Donald Duck", -200m),
    new("Ludwig von Drake", 20000m)
};
```

From the accounts collection, an immutable collection can be created with the extension method `ToImmutableList`. This extension method is available as soon as the namespace `System.Collections.Immutable` is opened.

```
ImmutableList<Account> immutableAccounts = accounts.ToImmutableList();
```

The variable `immutableAccounts` can be enumerated like other collections. It just cannot be changed:

```
foreach (var account in immutableAccounts)
{
  Console.WriteLine($"{account.Name} {account.Amount}");
}
```

Instead of using the `foreach` statement to iterate immutable lists, you can use the `ForEach` method that is defined with `ImmutableList<T>`. This method requires an `Action<T>` delegate as parameter, and thus a lambda expression can be assigned:

```
immutableAccounts.ForEach(a => Console.WriteLine($"{a.Name} {a.Amount}"));
```

When you work with these collections, methods like `Contains`, `FindAll`, `FindLast`, `IndexOf`, and others are available. Because these methods are like the methods from other collection classes discussed earlier in this chapter, they are not explicitly shown here.

In case you need to change the content for immutable collections, the collections offer methods such as `Add`, `AddRange`, `Remove`, `RemoveAt`, `RemoveRange`, `Replace`, and `Sort`. These methods are very different from normal collection classes because the immutable collection that is used to invoke the methods is never changed, but these methods return a new immutable collection.

Using Builders with Immutable Collections

Creating new immutable collections from existing ones can be done easily with the previously mentioned `Add`, `Remove`, and `Replace` methods. However, this is not very efficient if you need to do multiple changes that involve adding and removing many elements for the new collection. For creating new immutable collections that involve even more changes, you can create a builder.

Let's continue with the sample code and make multiple changes to the account objects in the collection. To do this, you can create a builder by invoking the `ToBuilder` method. This method returns a collection that you can change. In the sample code, all accounts with an amount larger than zero are removed. The original immutable collection is not changed. After the change with the builder is completed, a new immutable collection is created by invoking the `ToImmutable` method of the `Builder`. This collection is used next to output all overdrawn accounts (code file `ImmutableCollectionSample/Program.cs`):

```
ImmutableList<Account>.Builder builder = immutableAccounts.ToBuilder();
for (int i = builder.Count - 1; i >= 0; i--)
{
  Account a = builder[i];
  if (a.Amount > 0)
  {
    builder.Remove(a);
  }
}
ImmutableList<Account> overdrawnAccounts = builder.ToImmutable();
overdrawnAccounts.ForEach(a => Console.WriteLine(
  $"overdrawn: {a.Name} {a.Amount}"));
```

Other than removing elements with the `Remove` method, the `Builder` type offers the methods `Add`, `AddRange`, `Insert`, `RemoveAt`, `RemoveAll`, `Reverse`, and `Sort` to change the mutable collection. After finishing the mutable operations, invoke `ToImmutable` to get the immutable collection again.

Immutable Collection Types and Interfaces

Other than `ImmutableArray` and `ImmutableList`, the NuGet package `System.Collections.Immutable` offers some more immutable collection types as shown in the following table:

IMMUTABLE TYPE	DESCRIPTION
`ImmutableArray<T>`	`ImmutableArray<T>` is a struct that uses an array type internally but doesn't allow changes to the underlying type. This struct implements the interface `IImmutableList<T>`.
`ImmutableList<T>`	`ImmutableList<T>` uses a binary tree internally to map the objects and implements the interface `IImmutableList<T>`.
`ImmutableQueue<T>`	`ImmutableQueue<T>` implements the interface `IImmutableQueue<T>` that allows access to elements FIFO with `Enqueue`, `Dequeue`, and `Peek`.
`ImmutableStack<T>`	`ImmutableStack<T>` implements the interface `IImmutableStack<T>` that allows access to elements LIFO with `Push`, `Pop`, and `Peek`.
`ImmutableDictionary<TKey, TValue>`	`ImmutableDictionary<TKey, TValue>` is an immutable collection with unordered key/value pair elements implementing the interface `IImmutableDictionary<TKey, TValue>`.
`ImmutableSortedDictionary<TKey, TValue>`	`ImmutableSortedDictionary<TKey, TValue>` is an immutable collection with ordered key/value pair elements implementing the interface `IImmutableDictionary<TKey, TValue>`.
`ImmutableHashSet<T>`	`ImmutableHashSet<T>` is an immutable unordered hash set implementing the interface `IImmutableSet<T>`. This interface offers set functionality explained earlier in this chapter.
`ImmutableSortedSet<T>`	`ImmutableSortedSet<T>` is an immutable ordered set implementing the interface `IImmutableSet<T>`.

Like the normal collection classes, immutable collections implement interfaces as well—such as `IImmutableList<T>`, `IImmutableQueue<T>`, and `IImmutableStack<T>`. The big difference with these immutable interfaces is that all the methods that make a change in the collection return a new collection.

Using LINQ with Immutable Arrays

For using LINQ with immutable arrays, the class `ImmutableArrayExtensions` defines optimized versions for LINQ methods such as `Where`, `Aggregate`, `All`, `First`, `Last`, `Select`, and `SelectMany`. All that you need to use the optimized versions is to directly use the `ImmutableArray` type and open the `System.Linq` namespace.

The `Where` method defined with the `ImmutableArrayExtensions` type looks like this to extend the `ImmutableArray<T>` type:

```
public static IEnumerable<T> Where<T>(
  this ImmutableArray<T> immutableArray, Func<T, bool> predicate);
```

The normal LINQ extension method extends `IEnumerable<T>`. Because `ImmutableArray<T>` is a better match, the optimized version is used when you are invoking LINQ methods.

SUMMARY

This chapter took a look at working with different kinds of generic collections. Arrays are fixed in size, but you can use lists for dynamically growing collections. For accessing elements on a FIFO basis, there's a queue; and you can use a stack for LIFO operations. Linked lists allow for fast insertion and removal of elements but are slow for searching. With keys and values, you can use dictionaries, which are fast for searching and inserting elements. Sets are useful for unique items and can be ordered (`SortedSet<T>`) or not ordered (`HashSet<T>`).

Chapter 9 gives you details about working with arrays and collections by using LINQ syntax.